Hashing

What is it?

A form of narcotic intake?

A side order for your eggs?

A combination of the two?

Problem

- RT&T is a large phone company, and they want to provide enhanced caller ID capability:
  - given a phone number, return the caller's name
  - phone numbers are in the range 0 to $R = 10^{10}-1$
  - n is the number of phone numbers used
  - want to do this as efficiently as possible
- We know two ways to design this dictionary:
  - a ***balanced search tree*** (AVL, red-black) or a skip-list with the phone number as the key has O(log *n*) query time and O(*n*) space --- good space usage and search time, but can we reduce the search time to constant?
  - a ***bucket array*** indexed by the phone number has optimal O(1) query time, but there is a huge amount of wasted space: O(*n* + *R*)

(null)	(null)	...	Roberto	...	(null)
000-000-0000	000-000-0001	...	401-863-7639	...	999-999-9999

Another Solution

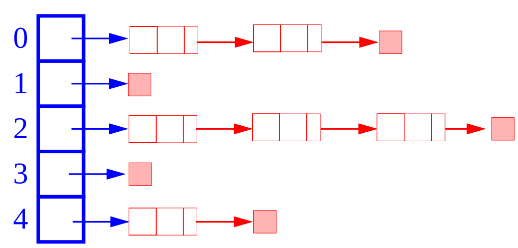
- A ***Hash Table*** is an alternative solution with O(1) expected query time and O(*n* + *N*) space, where *N* is the size of the table
- Like an array, but with a function to map the large range of keys into a smaller one
  - e.g., take the original key, ***mod*** the size of the table, and use that as an index
- Insert item (401-863-7639, Roberto) into a table of size 5
  - 4018637639 mod 5 = 4, so item (401-863-7639, Roberto) is stored in slot 4 of the table

				401-863-7639 Roberto
0	1	2	3	5

- A lookup uses the same process: map the key to an index, then check the array cell at that index
- Insert (401-863-9350, Andy)
- And insert (401-863-2234, Devin). We have a ***collision***!

Collision Resolution

- How to deal with two keys which map to the same cell of the array?
- Use ***chaining***
  - Set up ***lists*** of items with the same index



- The expected, search/insertion/removal time is O(*n*/*N*), provided the indices are uniformly distributed
- The performance of the data structure can be fine-tuned by changing the table size *N*

From Keys to Indices

- The mapping of keys to indices of a hash table is called a ***hash function***
- A hash function is usually the composition of two maps:
  - ***hash code map***: key → integer
  - ***compression map***: integer → $[0, N-1]$
- An essential requirement of the hash function is to map equal keys to equal indices
- A "good" hash function minimizes the probability of collisions
- Java provides a hashCode() method for the Object class, which typically returns the 32-bit memory address of the object.
- This default hash code would work poorly for Integer and String objects
- The hashCode() method should be suitably redefined by classes.

Popular Hash-Code Maps

- ***Integer cast***: for numeric types with 32 bits or less, we can reinterpret the bits of the nuber as an int
- ***Component sum***: for numeric types with more than 32 bits (e.g., long and double), we can add the 32-bit components.
- ***Polynomial accumulation***: for strings of a natural language, combine the character values (ASCII or Unicode) $a_0 a_1 \ldots a_{n-1}$ by viewing them as the coefficients of a polynomial:

$$a_0 + a_1 x + \ldots + x^{n-1} a_{n-1}$$

  - The polynomial is computed with ***Horner's rule***, ignoring overflows, at a fixed value x:

$$a_0 + x\,(a_1 + x\,(a_2 + \ldots\, x\,(a_{n-2} + x\, a_{n-1}) \ldots))$$

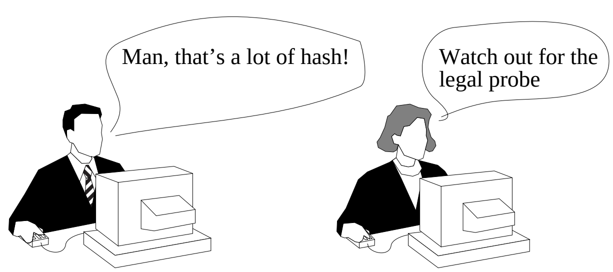
  - The choice $x = 33, 37, 39,$ or 41gives at most 6 collisions on a vocabulary of 50,000 English words
- Why is the component-sum hash code bad for strings?

Popular Compression Maps

- ***Division***: $h(k) = |k|$ **mod** N
  - the choice $N = 2^k$ is bad because not all the bits are taken into account
  - the table size N is usually chosen as a prime number
  - certain patterns in the hash codes are propagated
- ***Multiply, Add, and Divide*** (MAD): $h(k) = |ak + b|$ **mod** N
  - eliminates patterns provided a **mod** $N \neq 0$
  - same formula used in linear congruential (pseudo) random number generators

More on Collisions

- A key is mapped to an already occupied table location
  - what to do?!?
- Use a collision handling technique
- We've seen ***Chaining***
- Can also use ***Open Addressing***
  - ***Double Hashing***
  - ***Linear Probing***



Linear Probing

- If the current location is used, try the next table location

```
linear_probing_insert(K)
  if (table is full) error

  probe = h(K)

  while (table[probe] occupied)
        probe = (probe + 1) mod M

  table[probe] = K
```

- Lookups walk along table until the key or an empty slot is found
- Uses less memory than chaining
  - don't have to store all those links
- Slower than chaining
  - may have to walk along table for a long way
- Deletion is more complex
  - either mark the deleted slot
  - or fill in the slot by shifting some elements down

Linear Probing Example

- $h(k) = k \bmod 13$
- Insert keys:

18 41 22 44 59 32 31 73

0	1	2	3	4	5	6	7	8	9	10	11	12

Linear Probing Example (cont.)

		41			**18**	**44**	**59**	**32**	**22**	**31**	**73**	
0	1	2	3	4	5	6	7	8	9	10	11	12

Double Hashing

- Use two hash functions
- If M is prime, eventually will examine every position in the table

```
double_hash_insert(K)
  if(table is full) error

  probe = h1(K)
  offset = h2(K)

  while (table[probe] occupied)
    probe = (probe + offset) mod M

  table[probe] = K
```

- Many of same (dis)advantages as linear probing
- Distributes keys more uniformly than linear probing does

Double Hashing Example

- h1(K) = K mod 13
  h2(K) = 8 - K mod 8
  - we want h2 to be an offset to add

18 41 22 44 59 32 31 73

0	1	2	3	4	5	6	7	8	9	10	11	12

Double Hashing Example (cont.)

44		**41**	**73**		**18**	**32**	**59**	**31**	**22**			
0	1	2	3	4	5	6	7	8	9	10	11	12

Theoretical Results

- Let $\alpha = N/M$
  - the load factor: average number of keys per array index
- Analysis is probabilistic, rather than worst-case

Expected Number of Probes

	not found	***found***
Chaining	$1 + \alpha$	$1 + \frac{\alpha}{2}$
Linear Probing	$\frac{1}{2} + \frac{1}{2(1-\alpha)^2}$	$\frac{1}{2} + \frac{1}{2(1-\alpha)}$
Double Hashing	$\frac{1}{(1-\alpha)}$	$\frac{1}{\alpha} ln \frac{1}{1-\alpha}$

Expected Number of Probes vs. Load Factor